

W.Kluge

Abstract Computing Machines

A Lambda Calculus Perspective

With 89 Figures

 Springer

Authors

Prof. Dr. Werner Kluge
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Olshausenstrasse 40b, 24098 Kiel, Germany
wk@informatik.uni-kiel.de

Series Editors

Prof. Dr. Wilfried Brauer
Institut für Informatik der TUM
Boltzmannstrasse 3
85748 Garching
Germany
Brauer@informatik.tu-muenchen.de

Prof. Dr. Arto Salomaa
Turku Centre for Computer Science
Lemminkäisenkatu 14 A
20520 Turku
Finland
asalomaa@utu.fi

Prof. Dr. Grzegorz Rozenberg
Leiden Institute of Advanced Computer Science
University of Leiden
Niels Bohrweg 1
2333 CA Leiden
The Netherlands
rozenber@liacs.nl

Library of Congress Control Number: 2004117887

ACM Computing Classification (1998): D.3.2, D.3.4, F.3
ISBN 3-540-21146-2 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media
springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: KünkelLopka, Heidelberg
Typesetting: Camera ready by authors
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Printed on acid-free paper SPIN: 10991046 45/3142/YL - 5 4 3 2 1 0

Klaus Berkling

1931 – 1997

Preface

This monograph looks at computer organization from a strictly conceptual point of view to identify the very basic mechanisms and runtime structures necessary to perform algorithmically specified computations. It completely abstracts from concrete programming languages and machine architectures, taking the λ -calculus – a theory of computable functions – as the basic programming and program execution model. In its simplest form, the λ -calculus talks about expressions that are constructed from just three syntactical figures – variables, functions (in this context called abstractions) and applications (of operator to operand expressions) – and about a single transformation rule that governs the substitution of variable occurrences in expressions by other expressions. This β -reduction rule contains in a nutshell the whole story about computing, specifically about the role of variables and variable scoping in this game.

Different implementations of the β -reduction rule in conjunction with strategies that define the sequencing of β -reductions in complex expressions give rise to a variety of abstract λ -calculus machines that are studied in this text. These machines share, in one way or another, the components of Landin's SECD machine – a program text to be executed, a runtime environment that holds delayed substitutions, a value stack, and a dump stack for return continuations – but differ with respect to the internal representation of λ -expressions, specifically abstractions, the structure of the runtime environments and the mechanisms of program execution.

This text covers more than just implementations of functional or function-based languages such as MIRANDA, HASKELL, CLEAN, ML or SCHEME which realize what is called a weakly normalizing λ -calculus that uses a naive version of the β -reduction rule. The emphasis is instead on λ -calculus machines that are fully normalizing, using a complete and correct implementation of the β -reduction rule, which includes the orderly resolution of naming conflicts that may occur when free variables are substituted under abstractions. This feature is an essential prerequisite for correct symbolic computations that treat both functions and variables truly as first-class objects. It may, for instance, be

used to advantage in theorem provers to establish equality between two terms that contain variables, or to symbolically simplify expressions in the process of high-level program optimizations.

In weakly normalizing machines, the flavors of a full-fledged β -reduction are traded in for naive substitutions that are simpler to implement and require less complex runtime structures, resulting in improved runtime efficiency. Naming conflicts are consequently avoided by outlawing substitutions under abstractions, with the consequence that only ground terms (or basic values) can be computed. Weakly normalizing machines are therefore the standard vehicles for the implementation of functional or function-based languages whose semantics conform to this restriction. However, they are also used as integral parts of fully normalizing machines to perform the majority of those β -reductions that in fact can be carried out naively. Whenever substitutions need to be pushed under abstractions, a special mechanism equivalent to full β -reductions takes over to perform renaming operations that resolve potential name clashes.

Abstract machines for classical imperative languages are shown to be descendants of weakly normalizing machines that allow side-effecting operations, specified as assignments to bound variables, on the runtime environment. These side effects destroy important invariance properties of the λ -calculus that guarantee the determinacy of results irrespective of execution orders, leaving just the static scoping rules for bound variables intact. In this degenerate form of the λ -calculus, programs are primarily executed for their effects on the environment, as opposed to computing the values of the expressions of a weakly or fully normalizing λ -calculus.

This monograph, though not exactly mainstream, may be used in a graduate course on computer organization/architecture that focuses on the essentials of performing computations mechanically. It includes an introduction to the λ -calculus, specifically a nameless version suitable for machine implementation, and then continues to describe various fully and weakly normalizing λ -calculus machines at different levels of abstractions (direct interpretation, graph interpretation, execution of compiled code), followed by two kinds of abstract machines for imperative languages. The workings of these machines are specified by sets of state transition rules. The book also specifies, for code-executing abstract machines, compilation schemes that transform an applied λ -calculus taken as a reference source language to abstract machine code. Whenever deemed helpful, the execution of small example programs is also illustrated in a step-by-step fashion by sequences of machine state transitions.

I have used most of the material of this monograph in several graduate courses on computer organization which I taught over the years at the University of Kiel. Some of the material (Chaps. 2, 3 and the easier parts of Chaps. 4, 5) I even used in an undergraduate course on programming. The general impression was that at least the brighter students, after some time of getting used to the approach and to the notation, caught on pretty well to the message that I wanted to get across: understanding basic concepts and

principles of performing computations by machinery (with substitution as the most important operation) that are invariant against trendy ways of doing things in real computing machines, and how they relate to basic programming paradigms.

Acknowledgments

There are several people who contributed to this text with discussions and suggestions relating to its contents, with critical comments on earlier drafts, and with careful proofreading that uncovered many errors (of which some would have been somewhat embarrassing).

I am particularly indebted to Claus Reinke who gave Chaps. 5 to 8 and Appendix A a very thorough going-over, made some valuable recommendations that helped to improve verbal explanations and also the formal apparatus, specifically in Appendix A which I have largely adopted from his excellent PhD thesis, and provided me with a long list of ambiguities, notational inconsistencies and errors. Some intensive discussions with Sven-Bodo Scholz on head-order reduction, specifically on the problem of shared evaluation, led to substantial improvements of Chaps. 6 to 8. He also pointed out quite a few things in Chaps. 12 and 13 that needed clarification. I also had two enlightening discussions with Henk Barendregt and Rinus Plasmeijer on λ -calculus and on theorem proving which helped to shape Chaps. 4, 11 and Appendix B. Ulrich Bruening checked and made some helpful comments on Chaps. 13 and 14. Hans Langmaack was always available for some insightful discussions of language issues.

Makoto Amamiya gave me the opportunity to teach parts of this text in a one-week seminar course at Kyushu University in Fukuoka/Japan. The ensuing discussions gave me a fairly good idea of how the material would sink in with graduate students who have a slightly different background, and they also helped to correct a few flaws.

Kay Berkling, Claudia Schmittgen and Erich Valkema carefully proofread parts of a text that was more or less unfamiliar scientific territory to them, pointing out a few things that needed to be clarified, explained in more detail (by more examples), or simply corrected.

Last, not least, I wish to thank the people at Springer for their support of this project, especially Ingeborg Mayer, Ronan Nugent, Frank Holzwarth and, most importantly, Douglas Meekison who as a copyeditor did an excellent job of polishing the style of presentation, the layout of the text, and the English. There was hardly anything that escaped his attention.

... and there was Moni whose occasional peptalks kept me going.

Contents

1	Introduction	1
2	Algorithms and Programs	11
2.1	Simple Algorithms	14
2.1.1	Getting Started with Some Basics	15
2.1.2	Recursive Functions	19
2.1.3	The Termination Problem	23
2.1.4	Symbolic Computations	24
2.1.5	Operating on Lists	30
2.2	A Word on Typing	31
2.3	Summary	34
3	An Algorithmic Language	37
3.1	The Syntax of AL Expressions	38
3.2	The Evaluation of AL Expressions	41
3.3	Summary	47
4	The λ-Calculus	51
4.1	λ -Calculus Notation	52
4.2	β -Reduction and α -Conversion	53
4.3	An Indexing Scheme for Bound Variables **	59
4.4	The Nameless λ -Calculus	63
4.5	Reduction Sequences	68
4.6	Recursion in the λ -Calculus	73
4.7	A Brief Outline of an Applied λ -Calculus	78
4.8	Overview of a Typed λ -Calculus	79
4.8.1	Monomorphic Types	81
4.8.2	Polymorphic Types	83
4.9	Summary	86

5	The $SE(M)_{CD}$ Machine and Others	89
5.1	An Outline of the Original SECD Machine.	89
5.2	The $SE(M)_{CD}$ Machine	93
5.2.1	The Traversal Mechanism	94
5.2.2	Doing β -Reductions	96
5.2.3	Reducing a Simple Expression	98
5.3	The $\#SE(M)_{CD}$ Machine for the Nameless λ -Calculus	101
5.4	Implementing δ -Reductions	102
5.5	Other Weakly Normalizing Abstract Machines	105
5.5.1	The \mathcal{K} -Machine	105
5.5.2	The Categorical Abstract Machine	107
5.6	Summary	108
6	Toward Full-Fledged λ-Calculus Machines	113
6.1	Berkling's String Reduction Machine	115
6.2	Wadsworth's Graph Reduction Techniques	121
6.3	The $\lambda\sigma$ -Calculus Abstract Machine	125
6.3.1	The $\lambda\sigma$ -Calculus **	126
6.3.2	The Abstract Machine **	130
6.4	Head-Order Reduction	132
6.4.1	Head Forms and Head-Order β -Reductions	134
6.4.2	An Abstract Head-Order Reduction (HOR) Machine **	141
6.5	Summary	145
7	Interpreted Head-Order Graph Reduction	149
7.1	Graph Representation and Graph Reduction	150
7.2	Continuing with Reductions in the Head	156
7.3	Reducing the Tails	160
7.4	An Outline of the Formal Specification of G_{HOR}	163
7.5	Garbage Collection	164
7.6	Summary	167
8	The B-Machine	171
8.1	The Operating Principles of the B -Machine	173
8.2	The Instruction Set	174
8.2.1	Instruction Interpretation Without Sharing **	176
8.2.2	Interpretation Under Sharing in the Head **	179
8.3	Executing B -Machine Code: an Example **	181
8.4	Supporting Primitive Functions	186
8.5	Summary	189
9	The G-Machine	193
9.1	Basic Language Issues	195
9.2	Basic Operating Principles of the G -Machine	197
9.3	Compiling Supercombinators to G -Machine Code	201

9.4	<i>G</i> -Code for Primitive Functions	204
9.5	The Controlling Instructions *	205
9.6	Some <i>G</i> -Code Optimizations	209
9.7	Summary	211
10	The π-RED Machinery	215
10.1	The Basic Program Execution Cycle	215
10.2	The Operating Principles of the Abstract Machines	221
10.3	The Lazy Abstract Stack Machine LASM	223
10.3.1	The LASM Instruction Set	225
10.3.2	Compilation to LASM Code *	228
10.3.3	Some Simple Code Optimizations	232
10.4	The Strict Abstract Stack Machine SASM	235
10.4.1	The SASM Instruction Set	236
10.4.2	Compilation to SASM Code *	237
10.4.3	Code Execution	240
10.5	Reducing to Full Normal Forms *	244
10.6	Summary	249
11	Pattern Matching	253
11.1	Pattern Matching in AL	253
11.2	Programming with Pattern Matches	255
11.3	Preprocessing Pattern Matches	258
11.4	The Pattern Matching Machinery	260
11.5	Compiling Pattern Matches to LASM Code *	263
11.6	Code Generation and Execution: an Example **	265
11.7	Summary	268
12	Another Functional Abstract Machine	271
12.1	The Machine and How It Basically Works	272
12.1.1	Some Semantic Issues	273
12.1.2	Index Tuples and the Runtime Environment	275
12.2	The SECD _L Instruction Set	279
12.3	Compilation to SECD _L Code	282
12.4	Summary	285
13	Imperative Abstract Machines	289
13.1	Outline of an Imperative Kernel Language	291
13.2	An Example of an IL Program	294
13.3	The Runtime Environment	296
13.3.1	Using Static and Dynamic Links	298
13.3.2	Dropping Dynamic Links	300
13.3.3	Calculating Stack Addresses *	302
13.4	The Instruction Set	304
13.5	Compiling IL Programs to IAM Code *	306

13.6	Compiling the Bubble-Sort Program	309
13.7	Outline of a Machine for a ‘Flat’ Language	312
13.8	Summary	318
14	Real Computing Machines	321
14.1	A Typical CISC Architecture	323
14.1.1	The Register Set, Formats and Addressing in Memory	324
14.1.2	Addressing Modes	326
14.1.3	Some Important Instructions	328
14.1.4	Implementing Procedure Calls	330
14.2	A Typical RISC Architecture	334
14.2.1	The SPARC Register Set	335
14.2.2	Some Important SPARC Instructions	339
14.2.3	The SPARC Assembler Code for Factorial	341
14.3	Summary	344
A	Input/Output	347
A.1	Functions as Input/Output Mappings	348
A.2	Continuation-Style Input/Output	354
A.3	Interactions with a File System	357
B	On Theorem Proving	361
	References	369
	Index	377